IT UNIVERSITY OF COPENHAGEN

A Comparative Study on Matrix Multiplication Algorithms

Katarzyna Toborek and Gustav Gyrst

Applied Algorithms Exam Report
MSc in Software Design
Katarzyna Toborek and Gustav Gyrst
ktob@itu.dk & gugy@itu.dk

December 17, 2021

CONTENTS

1 THE ALGORITHMS

1.1 INTRODUCTION

Matrix multiplication is the mathematical operation of multiplying two matrices. It is a fundamental linear algebra tool that is applied in several domains[1]. The following paper presents different algorithms to solve matrix multiplication of square matrices with side lengths defined as $n = 2^x$ with $x > 0$. In particular, we present and run experiments to compare the performance measured in time of six different matrix multiplication algorithms (see appendix A.3.2 for reference). All code for this exam report is written in Python.

1.2 IMPLEMENTATIONS

For the implementation of the six algorithms we use the *Matrix* class provided in the code template handed out for the purpose of this report. The *Matrix* class is a simple c-order (i.e., row-major) matrix class with a *Numpy* backend. It uses 64bit floating point numbers (numpy documentation, 2021). While some methods in the *Matrix* class were already defined, others have been implemented. In particular, we have implemented add and subtract methods. For each we have made in-place and ordinary implementations[2]. For the in-place subtraction and addition methods we add and subtract to an existing variable, whereas the ordinary add and subtract return a new matrix instance with the result.

1.2.1 Nested Loops

The implementations of the algorithms follow the structures presented in class and materials given. First, all algorithms were implemented without the Matrix class, then they were modified to fit the Matrix class. Small optimization choices have been made to avoid the relative expensive function calls in Python (python documentation, 2021). In particular, we have used the method *._arr* directly instead of calling the *.__getitem__* in the places where it was possible (see A.3.2) as *.__getitem__* is frequently called in our algorithm implementations.

First, the **elementary** algorithm (*elementary_multiplication*) is a simple algorithm based on three nested loops. Our implementation of the algorithm follows the straight forward ijk ordering for the loops.

Second, the **transposed** algorithm (*elementary_multiplication_transposed*) is an alternative implementation of the elementary algorithm. Recall, in matrix multiplication we need to treat each row of matrix A and column of matrix B to get the product Matrix C. The key words here are *row* and *column*. With the use of a helper method *transpose*, this variant assumes the second matrix transposed so that both can be read in row-order sequentially. Assuming C-order processing of the data, one would assume a performance gain over the *elementary* algorithm.

Third, the **tiled** algorithm (*tiled_multiplication*) is based on the *elementary* variant. It however divides the matrices into sub-matrices determined by the parameter *s* that it takes as input, and then attacks each sub-matrix with the elementary algorithm. The purpose of dividing the problem into sub problems is to keep the matrices within the computer's cache.

[1] Matrix multiplication is applied in computer science, engineering, physics, and more.
[2] In-place refers to operations -= and +=, whereas the ordinary simply is + and -.

1.2.2 Recursion

Fourth, the recursive **copying** algorithm (*recursive_multiplication_copying*) computes the matrix multiplication with recursion. In particular it makes eight recursive calls, diving into the sub-matrices until it reaches the base-case where n=1 (i.e., a matrix cell is reached). At each recursion level it copies a new instance of the product matrix C, hence the name *copying* variant. This can be argued to be fairly inefficient.

Fifth, an alternative approach is the recursive **write through** algorithm (*recursive_multiplication_write_through*). Contrary to the *copying* variant, we now take the product matrix C as a parameter which it then recurses down with in the sub-matrices.

Sixth, the last implementation is **Strassen's** algorithm (*strassen*). This is an interesting algorithm as it only has 7 recursive sub problems, which gives it a better asymptotic runtime, than the earlier presented algorithms in this paper; Strassen namely has $\mathcal{O}(n^{2.8})$. It does however have more arithmetic operations than the other versions (see A.3.2).

1.3 TESTING FOR CORRECTNESS

Firs, the algorithms were tested in Codejudge[3] when they were first implemented. Thereafter, all algorithms went through several tests. The first and most important was to test the input that we ran our experiments on, namely positive integers x in the format of 64bit floats determined as follows:

$$x < \sqrt{2^{53}/n}$$

The seed functionality was tested by running the input generation multiple with a specific seed.

Additional tests with various input types were conducted for each implementation. As a default, the above described floats had no decimals unless specified otherwise. The tested inputs for matrices are as follows: positive floats, positive long floats (floats with long decimals), positive and negative floats, 0's and 1's and positive floats larger than the safe limit described in the formula presented above. Each implementation was tested on matrix sizes between 2-512 (with n always being a power of 2) The output produced by the implemented algorithms was compared using `numpy.testing.assert_equal` against an expected output, which was produced with the `numpy.matmul` method. As expected, all algorithms failed the tests with long floats input, due to the floating point rounding error. To ensure that this was indeed the case, the resulting output was compared against the expected one using `numpy.testing.assert_allclose` method, which asserts two inputs as equal up to desired tolerance, with default relative tolerance equal 1e-07. Tests with over-the-limit floats were generally failing with given power of 2 > 59.

Strassen's algorithm was a particular case. The positive floats tests with `assert_equal` passed only for n=2 and n=4. This could be due to the algorithm containing many arithmetic operations, which might make it more prone to floating point rounding errors. The remaining matrix sizes passed the tests using the method `assert_allclose`. This meant that the safe threshold of $x < \sqrt{2^{53}/n}$ doesn't hold for that implementation. Power of 2 > 48 already resulted in the small imprecisions occurring as shown in Table 10 in A.1.5.

The implementations were furthermore tested with input where failure was expected, such as non-square matrices or for the tiled algorithm with parameter $s > n$.

[3] Codejudge is an online code tester. There were Codejudge tests for the elementary and recursive copying variant.

2 EXPERIMENTS

2.1 THE SET UP

The experiments were conducted on a Surface book 3 with 1.30GHz 8 cores Intel i7 processor with 32GiB RAMS and cache as specified in A.1.6. The machine runs the linux distribution POP_OS v20.04 (see A.1.7) and all experiments were run from the terminal. We tried to run the experiments in isolation to the best extent possible, however, as we run it on a modern multitasking operating system, evidently, some processes run in the background. We used anaconda Python 3.8 for all experiments. We used the python *random* module to generate our random 64bit floats and the *time* module to make our time measurement (see A.3.1). In particular, we ran experiments with a seed to ensure full reproducibility for all experiments. For experiments with increasing n, we created the seed with the specific n value in order to get the same matrices for each algorithm to ensure comparability. In general, all experiments were run with two warm-up runs of the algorithms[1]. Then three runs with measurements, where the average of these where presented as the result. We decided not to include error bars for standard deviation as the variance in our results were negligible and this influenced the readability of the graphs. For further details, a complete summary of the set-up is provided in Table 13 in A.1.8.

2.2 S FOR TILED

To find the optimal s for our tiled algorithm we ran experiments trying all the different s with fixed n sizes[2] of $n = 128$ (i.e., $n = 2^7$ and $n_p = 7$) and $n = 256$ (i.e., $n = 2^8$ and $n_p = 8$). We thus run the experiments with values 2^x where:

$$2^1 \leqslant 2^x \leqslant 2^{n_p - 1}$$

Our results presented in Figure 2 in A.2.1 show us that as we increase s, the performance increases until we reach $s = 32$. Hereafter the performance gain stagnates as we increase s further. Thus we find $s = 32$ to be the ideal size for s. When $s = 32$ it corresponds to dividing the matrix of size 256 into 8 sub problems. This result is somewhat expectable as the smaller the sub-problem, the more likely it is for the machine to fit the sub components in the cache (i.e., L1, L2, L3). Given that the cache of the machine we are running the experiments on has size = 1,049,000 (see A.1.6 for reference), we would have to run the experiment on larger matrices in order to have a scenario where the machine has to fetch data in *memory*. Theoretically, the matrix side length n has to be set the following way for it to exceed the cache size:

$$3 * (n^2 * 8) > cache\ size \qquad [3]$$

This would mean that we would need matrices with $n = 512$ in order to start exceeding the cache. We do however already see gains in performance on this smaller experiment with n=256. This could be related to gains from matrices fitting into the sub levels in cache.

[1] This was done to ensure that the cache was warmed up.
[2] I.e., we run the experiments with multiples of 2, from 2-128 for example if $n = 256$.
[3] We multiply with 3 because we have three matrices A,B and the product matrix C. We multiply with 8 as we want to represent our result in Bytes. Recall the matrices are build on 64bit floats.

2.3 M FOR WRITE THROUGH AND STRASSEN

To optimize the write through and Strassen's algorithm we introduce a variable m. when n<=m the algorithms will switch to the elementary matrix multiplication approach. We thus ran experiments to determine the ideal m values for the recursive write through and Strassen's algorithm. This was done with some simple experiments where we kept a fixed n (i.e., n=256), and then increase the m value in similar manner as in the experiment in section 2.2. We, however, include 0 in our set of m values, so that we also see the performance when it never swaps. As described in the figures presented in A.2.2 we find that the optimal m for the recursive write through to be 16[4]. For Strassen we find the optimal m to be 8. We see a similar picture when testing with different n values (see Figure 5 and 6 in A.2.2).

2.4 THE HORSE RACE

With our findings of s and m values from sections 2.2 and 2.3, we run the horse race to compare the performance of all our algorithms. The results are presented below in Figure 1 with log scaling on both axes and the time presented in $\log(\mathrm{nano\_s}/n^3)$. The copying variant comes out as the worst performing algorithm. This is no surprise as recursion is expensive in python and the algorithm makes a full copy of the product matrix C at each recursion level. Thus it has a lot of overhead compared to the write through that performs relatively better. The tiled comes out as the second worst algorithm in our run[5]. This bad performance could be related to the small n size we run our experiments on. The remaining algorithms all have very similar performance. However, interestingly we see that for the larger n values for Strassen's algorithm seems to triumph over the others. This could be explained by its superiority in terms of asymptotic run-time.

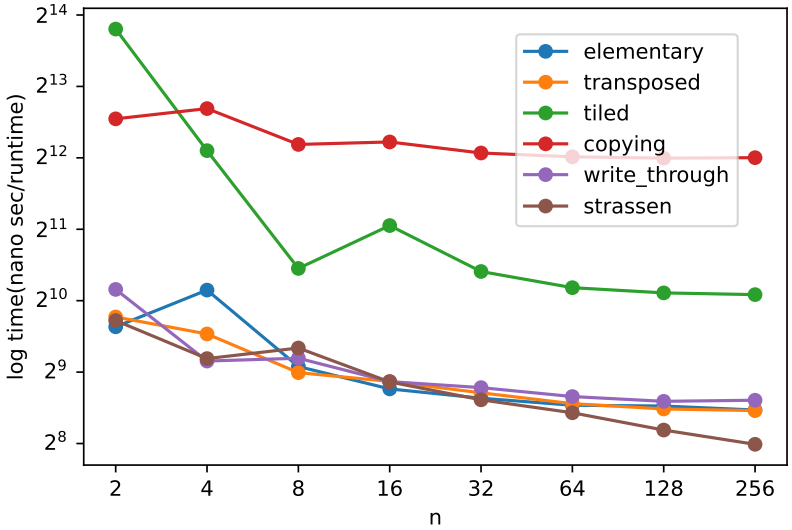


Figure 1: Horse race results log-scaled on both axis. n presented on the x-axis and $\log(\mathrm{nano\_s}/n^3)$ on y. Thus when a line is descending we see faster performance than cubic. $s_{\mathrm{tiled}} = 32$, $m_{\mathrm{write\_through}} = 16$, $m_{\mathrm{strassen}} = 8$.

[4] The curve stagnates from $n = 16$ and forward. Thus we have similar performance for these values. We decided to take 16, so that it would not be running the elementary algorithm for the most part (see A.2.2).

[5] The fluctuation for the small n values is related to the fact that we divide by 8 (i.e., the ideal s=32 when n=256). This we cannot do for the small numbers. Instead we divide with 2. See the *benchmark* method in `tiled_matrix_multiplication_e.py` in the code attached.

3 DISCUSSION

3.1 RESULTS & LIMITATIONS

While our results do show differences in the performance of the algorithms, our experiments would most likely have benefited from running on larger matrices. However, with approximately cubic growth in run-time of the algorithms and the current inefficient implementations of the algorithms with the *Matrix* class and Cpython, it would take time beyond what is possible with the resources available for running these experiments. This is how we justify our choice of n. Better and faster implementations would perhaps have given us more interesting findings. In relation to the tiled algorithm, it might have enabled us to run experiments of larger size and see the benefits of tiling when the alternative is to fetch data in memory. In particular, for the *tiled* experiment presented in section 2.2 experiment, we would expect the curve to have a U shape when running on larger n sizes, where we would see the most efficient s parameter to be approximately when the three matrices could fit inside the cache (with some buffer space for noise from other processes running the machine in background). This we could not see with our current implementation and experiments.

Another somewhat surprising result is that we do not see any performance gain in the elementary *transposed* variant over the normal elementary algorithm. And this is even when the transposing happens before the measurement. Again this could be related to the current relatively small size of our experiment with $n = 256$. However it is also likely to be explained by the implementation or possibly the backend, that prevent the algorithm from benefiting from the sequential c-order reading of the two matrices. The optimal m values we found for the recursive write through show us that in general the algorithm is better of when going over to the elementary approach at a given point. This makes sense as there is quite some overhead associated with recursion. Interestingly, Strassen's algorithm that only has 7 recursive subproblems, we also see a lower optimal swap value m. This aligns with our assumption of recursion having a high overhead. We also ran the experiments setting $m = 0$ so that the recursive write through and Strassen's algorithm never fall back to the elementary variant (see Figure 7 in A.2.3). This shows us a clearer picture of the overhead associated with recursion as the elementary nested loop approach then perform best in our experiments.

3.2 FUTURE IMPROVEMENTS

To gain better insights and a clearer picture for our results, we see two obvious steps for improvement. 1) running the experiments on larger matrices, 2) redo the project without the use of the *Matrix* class (and hence not Cpython and the Numpy backend), but instead implementing everything running with Pypy on more basic list structures (i.e., using the list template provided for this experiment). Ideally, we would run the experiment in a more low-level languages such as C. This could likely solve the issue of the excess overhead from function calls (python documentation, 2021) and general costs of running experiments in a high-level language like Python.

BIBLIOGRAPHY

numpy documentation (2021). numpy.finfo. https://numpy.org/doc/stable/reference/generated/numpy.finfo.html (visited on 15.12.2021),.

python documentation (2021). performance-tips. https://wiki.python.org/moin/PythonSpeed/PerformanceTips (visited on 15.12.2021),.

A APPENDIX

A.1 TABLES

A.1.1 Results for running the tiled algorithm with n=256 to find the optimal s value

Table 1: Tiled multiplication for n=256 with different values of s

s	time	stdv	nano_seconds/runtime
2	35.286822	0.263627	2103.258500
4	25.406195	0.004774	1514.327219
8	22.457708	0.027359	1338.583691
16	21.006982	0.028259	1252.113691
32	21.064733	0.052177	1255.555925
64	21.161859	0.075981	1261.345095
128	21.150426	0.082496	1260.663652

A.1.2 Results for running the write_through algorithm with n=256 to find the optimal m value

Table 2: Write_through algorithm for n=256 with different values of m

m	time(s)	stdv	nano_seconds/runtime
0	85.050945	0.570346	5069.431386
2	17.109285	0.128721	1019.792876
4	8.851750	0.067093	527.605389
8	7.246603	0.039321	431.931217
16	6.829965	0.067456	407.097649
32	6.673965	0.006570	397.799312
64	6.697631	0.042814	399.209938
128	6.634563	0.020842	395.450778

A.1.3 Results for running the Strassen's algorithm with n=256 to find the optimal m value

Table 3: Strassen algorithm for n=256 with different values of m

m	time(s)	stdv	nano_seconds/runtime
0	50.291963	0.017154	9087.128834
2	12.671518	0.020420	2289.584876
4	4.802607	0.011825	867.771025
8	3.826228	0.010634	691.351636
16	3.862172	0.014518	697.846207
32	4.313387	0.019903	779.375063
64	4.935135	0.028239	891.717247
128	5.584180	0.021938	1008.991417

A.1.4 Results of the horse race for each algorithm implementation

Table 4: Elementary multiplication horse race results

n	time	stdv	nano_seconds/runtime
2	0.000006	3.641904e-07	794.728597
4	0.000073	3.779166e-05	1134.971778
8	0.000277	1.740673e-05	540.477534
16	0.001781	1.551248e-04	434.927642
32	0.013024	3.113044e-04	397.458886
64	0.097294	8.575994e-04	371.147811
128	0.772442	2.838363e-02	368.328983
256	5.945595	8.985034e-03	354.385084

Table 5: Elementary transposed multiplication race results

n	time	stdv	nano_seconds/runtime
2	0.000007	7.664085e-07	874.201457
4	0.000047	2.175365e-05	741.332769
8	0.000261	3.251946e-05	509.433448
6	0.001915	3.008783e-04	467.640348
32	0.013720	3.087911e-04	418.690130
64	0.098792	2.661016e-03	376.861256
128	0.750998	4.252302e-03	358.103951
256	5.913399	4.083528e-02	352.466046

Table 6: Tiled multiplication race results with s=32

n	time	stdv	nano_seconds/runtime
2	0.000114	0.000024	14295.180639
4	0.000281	0.000093	4392.117262
8	0.000717	0.000067	1400.243491
16	0.008696	0.000141	2123.124432
32	0.044537	0.000432	1359.175561
64	0.304559	0.000336	1161.799446
128	2.316262	0.003941	1104.479982
256	18.219651	0.013749	1085.975844

Table 7: Recursive copying multiplication race results

n	time	stdv	nano_seconds/runtime
2	0.000048	0.000024	5980.332692
4	0.000423	0.000037	6607.423226
8	0.002388	0.000379	4664.994776
16	0.019572	0.000537	4778.344495
32	0.140677	0.001218	4293.125433
64	1.084478	0.002988	4136.953976
128	8.562554	0.010659	4082.944088
256	68.827789	0.293615	4102.455885

Table 8: Recursive write_through multiplication race results with m=16

n	time	stdv	nano_seconds/runtime
2	0.000009	4.145559e-06	1142.422358
4	0.000036	2.384186e-07	569.969416
8	0.000300	1.714792e-05	585.801899
16	0.001913	4.276741e-04	466.941856
32	0.014431	2.937027e-04	440.411289
64	0.105869	1.890130e-03	403.856878
128	0.808589	7.439260e-03	385.565196
256	6.537513	5.919515e-02	389.666151

Table 9: Strassen multiplication race results with m=8

n	time	stdv	nano_seconds/runtime
2	0.000007	3.641904e-07	844.399134
4	0.000037	3.641904e-07	583.628813
8	0.000331	8.036982e-06	646.958748
16	0.001906	1.670243e-04	465.428457
32	0.012821	5.661848e-04	391.267046
64	0.090449	4.253191e-04	345.037430
128	0.611892	2.409683e-04	291.772722
256	4.262170	3.451533e-03	254.045148

A.1.5 Testing Strassen on Large Numbers

power	% fail rate
47	0.0
50	0.018
53	0.61
56	0.90
59	0.92
62	0.92
65	0.92
68	0.92
71	0.92
74	0.93
77	0.93

Table 10: The power specified above is the power of the inputs we are generating. recall the formula: $x < \sqrt{2^{53}/n}$ from section 1.3. The power where we expect the full accuracy for our 64bit floats is 53. We see above that strassen fails earlier. It starts getting issues already when the power=50.

A.1.6 Cache of Surface Book 3 (2020 model)

Cache Level	Size	
L1 (Data)	192 KiB	24,576 Bytes
L1 (Instruction)	128 KiB	16384 Bytes
L2	2 MiB	262,144 Bytes
L3	8 MiB	1,049,000 Bytes

Table 11: The Cache sizes of the computer that our experiments were run on; namely a Surface Book 3 (2020 model)

A.1.7 OS Details for Machine

Machine OS Details	
Operating System	Pop!_OS 20.04 LTS
Kernel	Linux 5.14.15-surface
Architecture	x86-64

Table 12: The OS Details of the Machine the experiments were run on.

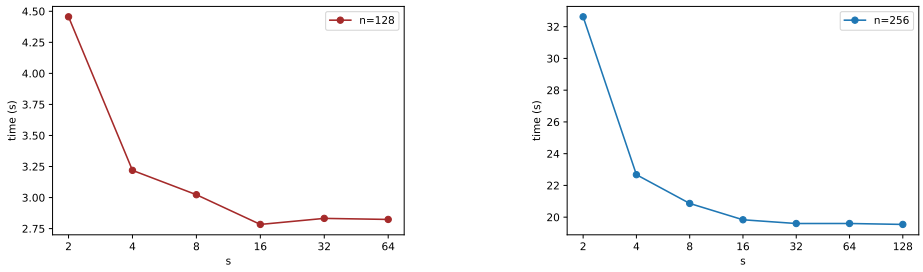
A.1.8 Experiment Set Up Details

Title	Description
Benchmarking	Our benchmarking methods generally take the shape of two nested loops. First, one that iterates over the different n (or m & s values) that we wish to run the experiment on. Here, the random inputs are generated and two warm-up runs are made. Next, there is another loop that iterates over N repetition. That is the number of time we run the experiment. The result returned is then converted into a CSV-file, Pandas Dataframe, and lastly the plots. (See `Plotting.ipynb` in the code attached for more details).
Measurement	In `measurement.py` we have our *measure* method. It is simple method taking and returning the time in seconds that it took for an experiment to run. It is based on the python module *time*.
Input Generator	Our input generator follows the specifications in the *instruction* provided with the exam assignment. It creates integers x (i.e., whole numbers) within a dynamic range that decrease when n increases. $$x < \sqrt{2^{53}/n}$$ See A.3.1 in the A.3.2 (code) section of the appendices for more detail.

Table 13: Above a more detailed description of the experiment set up is presented.

A.2 FIGURES

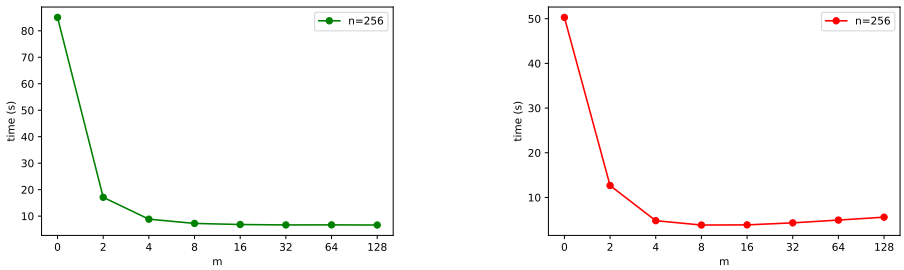
A.2.1 The optimal S for the Tiled algorithm



(a) n=128 **(b)** n=256

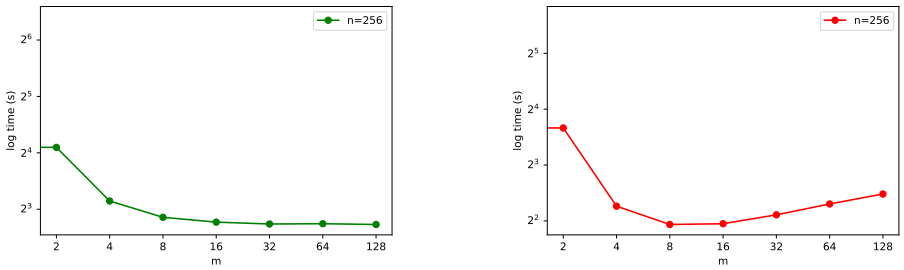
Figure 2: Results from the tiled experiment where we test different s values on a fixed n. (a) shows the experiment with n=128, and (b) shows the experiment with n=256.

A.2.2 M Value for Write Through and Strassen



(a) write_through n=256 **(b)** strassen n=256

Figure 3: Results from recursive write_through and strassen experiment to find the optimal m value.



(a) write_through n=256 **(b)** strassen n=256

Figure 4: Result from recursive write_through and Strassen experiment with log scale applied on both axes.

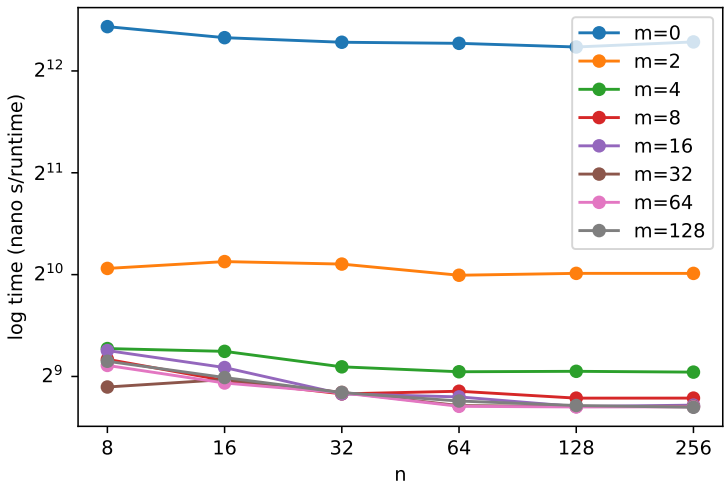


Figure 5: m results for write through log-scaled on both axes. n presented on the x-axis and time (in nanoseconds) / n^3 on the y-axis.

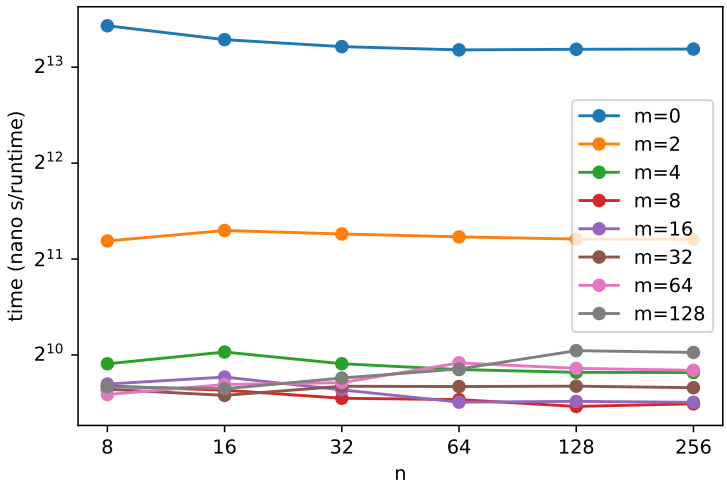


Figure 6: m results for Strassen log-scaled on both axes. n presented on the x-axis and time (in nanoseconds) / $n^{2.8}$ on the y-axis.

A.2.3 Horse Race with m=0 for Write Through and Strassen

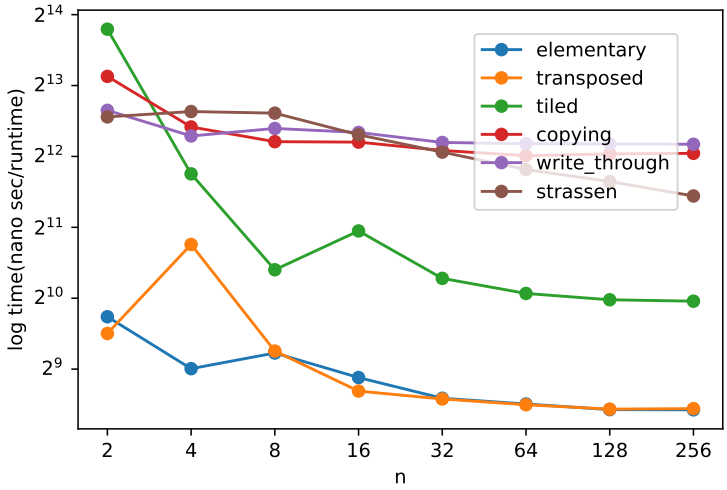


Figure 7: Horse race results log-scaled on both axis. n presented on the x-axis and time (in nanoseconds) / n^3. Thus when a line is descending we see faster performance than cubic. $s_{tiled} = 32$, $m_{write\_through} = 0$, $m_{strassen} = 0$.

A.2.4 Horse Race (other visualizations)

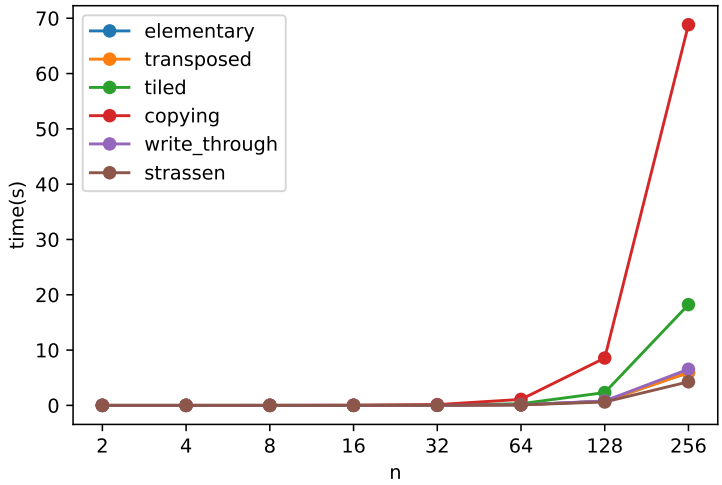


Figure 8: Horse race results log scaled x-axis. $s_{tiled} = 32$, $m_{write\_through} = 16$, $m_{strassen} = 8$.

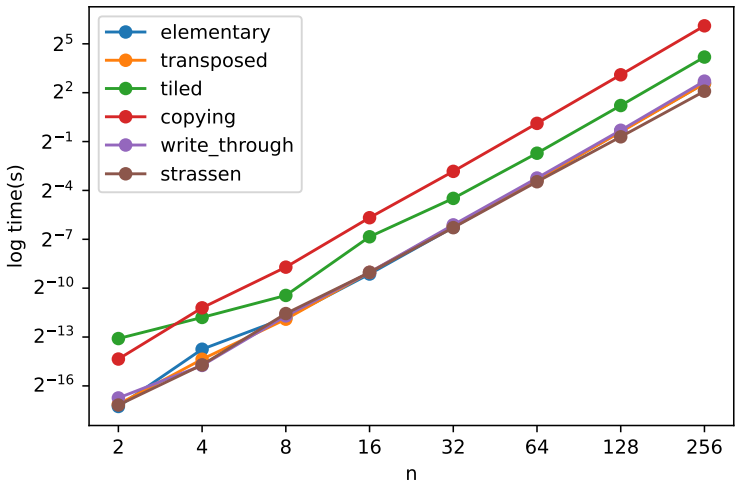


Figure 9: Horse race results log scaled x-axis and y-axis. $s_{tiled} = 32$, $m_{write\_through} = 16$, $m_{strassen} = 8$.

A.3 CODE

A.3.1 Measurements & Input Generation

```
def measure(f: Callable [[],Any])->float:
    start: float = time.time()
    f()
    end: float = time.time()
    return end - start
```

Python code A.1: *measure* used for measuring the time in the experiments

```
1
2  def get_input_range(n):
3      lower_bound = 0
4      upper_bound = round(np.sqrt(2**(53)/n))
5      input_range = [lower_bound, upper_bound]
6      return input_range
7
8
9  def generate_input(n: int) -> Matrix :
10     list= []
11     input_range = get_input_range(n)
12     for i in range(0,n*n):
13         random.seed(n + i)
14         l = random.randint(input_range[0],int(input_range[1]))
15         list.append(float(l))
16     return Matrix(n,n,np.array(list).reshape(n,n))
```

Python code A.2: *generate_input* used to generate random inputs in the range defined so that the experiments don't run into floating point errors. We are working with Numpy's 64bit floats and have selected range upper_bound thereafter so it dynamically scales with n.

A.3.2 Matrix Multiplication Algorithms

```
1  def elementary_multiplication(A: Matrix, B: Matrix)->Matrix:
2      n = A.cols()
3      C = Matrix(n,n)
4      for i in range(n):
5          for j in range(n):
6              for k in range(n):
7                  C._arr[i,j] += A._arr[i,k]*B._arr[k,j]
8      return C
```

Python code A.3: elementary_multiplication

```
1  def transpose(A: Matrix)->None:
2      a: int = A.cols()
3      b = 0
4      for i in range(0,a):
5          for j in range(b,a):
6              if(i != j):
7                  t = A._arr[i,j]
8                  A._arr[i,j] = A._arr[j,i]
9                  A._arr[j,i] = t
10         b += 1
11
12 def elementary_multiplication_transposed(A: Matrix, B: Matrix)->Matrix:
13     n = A.cols()
14
15     C = Matrix(n,n)
16     for i in range(n):
17         for j in range(n):
18             for k in range(n):
19                 C._arr[i,j] += A._arr[i,k]*B._arr[j,k]
20
21     return C
```

Python code A.4: elementary_multiplication_transposed

```
1  def tiled_multiplication(A: Matrix, B: Matrix, s: int)->Matrix:
2      n = A.cols()
3      C = Matrix(n,n)
4
```

```
    for i in range(n//s):
        for j in range(n//s):
            for k in range(n//s):
                subA = A[i*s:i*s+s,k*s:k*s+s]
                subB = B[k*s:k*s+s,j*s:j*s+s]
                z = subA.cols()
                subC = Matrix(z,z)
                for l in range(z):
                    for m in range(z):
                        for o in range(z):
                            temp = subC[l,m] + subA[l,o]*subB[o,m]
                            subC[l,m] = temp
                C[i*s:i*s+s,j*s:j*s+s].__iadd__(subC)

    return C
```

Python code A.5: tiled_multiplication

```

def recursive_multiplication_copying(A:Matrix , B:Matrix) -> Matrix:

    n = A.rows()

    if A.rows().__eq__(1):

        C = A[0]*B[0]
        return C

    else:
        C = Matrix(A.rows(), A.cols())

        C00 = C[:n//2,:n//2]
        C01 = C[:n//2,n//2:]
        C10 = C[n//2:,:n//2]
        C11 = C[n//2:,n//2:]


        P0 = A[:n//2,:n//2]
        P1 = A[:n//2,n//2:]
        P2 = A[:n//2,:n//2]
        P3 = A[:n//2,n//2:]
        P4 = A[n//2:,:n//2]
        P5 = A[n//2:,n//2:]
        P6 = A[n//2:,:n//2]
        P7 = A[n//2:,n//2:]

        Q0 = B[:n//2,:n//2]
        Q1 = B[n//2:,:n//2]
        Q2 = B[:n//2,n//2:]
        Q3 = B[n//2:,n//2:]
        Q4 = B[:n//2,:n//2]
        Q5 = B[n//2:,:n//2]
        Q6 = B[:n//2,n//2:]
        Q7 = B[n//2:,n//2:]

        M0 = recursive_multiplication_copying(P0, Q0)
        M1 = recursive_multiplication_copying(P1, Q1)
        M2 = recursive_multiplication_copying(P2, Q2)
        M3 = recursive_multiplication_copying(P3, Q3)
        M4 = recursive_multiplication_copying(P4, Q4)
        M5 = recursive_multiplication_copying(P5, Q5)
        M6 = recursive_multiplication_copying(P6, Q6)
        M7 = recursive_multiplication_copying(P7, Q7)

        C00 += M0 + M1
```

```
        C01 += M2 + M3
        C10 += M4 + M5
        C11 += M6 + M7

        return C
```

Python code A.6: recursive_copying_multiplication. Implementation follows exactly the structure as provided in the template `Matrix.py` file provided for this exam assignment as the instruction for the task demands.

```

def recursive_multiplication_write_through(A: Matrix, B: Matrix, C:Matrix, m=0):

    #initializing C and getting the length of n
    n = A.rows()

    if n <= m:
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    C._arr[i,j] += A._arr[i,k] * B._arr[k,j]
        return C

    elif n == 1:
        C._arr[0] += A._arr[0]*B._arr[0]
        return C

    else:
        #M0 C upper left    a00             b00             c00
        a00b00 = r(A[:n//2,:n//2], B[:n//2,:n//2], C[:n//2,:n//2], m)
        #M1 C upper left    a01             b10             c00
        a01b10 = r(A[:n//2,n//2:], B[n//2:,:n//2], C[:n//2,:n//2], m)

        #M2 C upper right   a00             b01             c01
        a00b01 = r(A[:n//2,:n//2], B[:n//2,n//2:], C[:n//2,n//2:], m)
        #M3 C upper right   a01             b11             c01
        a01b11 = r(A[:n//2,n//2:], B[n//2:,n//2:], C[:n//2,n//2:], m)

        #M4 C lower left    a10             b00             c10
        a10b00 = r(A[n//2:,:n//2], B[:n//2,:n//2], C[n//2:,:n//2], m)
        #M5 C lower left    a11             b10             c10
        a11b10 = r(A[n//2:,n//2:], B[n//2:,:n//2], C[n//2:,:n//2], m)

        #M6 C lower right   a10             b01             c11
        a10b01 = r(A[n//2:,:n//2], B[:n//2,n//2:], C[n//2:,n//2:], m)
        #M7 C lower right   a11             b11             c11
        a11b11 = r(A[n//2:,n//2:], B[n//2:,n//2:], C[n//2:,n//2:], m)

        return C
```

Python code A.7: recursive_multiplication_write_through. Note that r is a mere replacement of the long title and only added for better readability in the code. The actual code has the full name so it call itself recursively as intended.

```
def strassen(A: Matrix, B: Matrix, m=0)->Matrix:

    n = A.rows()

    if n <= m:
        C = Matrix(n,n)
        for i in range(n):
            for j in range(n):
                for k in range(n):
                    C._arr[i,j] += A._arr[i,k]*B._arr[k,j]
        return C
```

```
    elif n == 1:
        C = A._arr[0]*B._arr[0]
        return C

    else:
        #C sized to A at the particular recursion level
        C = Matrix(A.rows(), A.rows())

        #Cutting our copy of C into four sections
        C00 = C[:n//2,:n//2]
        C01 = C[:n//2,n//2:]
        C10 = C[n//2:,:n//2]
        C11 = C[n//2:,n//2:]

        # P1 = A00 + A11
        P1 = A[:n//2,:n//2] + A[n//2:,n//2:]
        # P2 = A10 + A11
        P2 = A[n//2:,:n//2] + A[n//2:,n//2:]
        # P3 = A00
        P3 = A[:n//2,:n//2]
        # P4 = A11
        P4 = A[n//2:,n//2:]
        # P5 = A00 + A01
        P5 = A[:n//2,:n//2] + A[:n//2,n//2:]
        # P6 = A10 - A00
        P6 = A[n//2:,:n//2] - A[:n//2,:n//2]
        # P7 = A01 - A11
        P7 = A[:n//2,n//2:] - A[n//2:,n//2:]

        # Q1 = B00 + B11
        Q1 = B[:n//2,:n//2] + B[n//2:,n//2:]
        # Q2 = B00
        Q2 = B[:n//2,:n//2]
        # Q3 = B01 - B11
        Q3 = B[:n//2,n//2:] - B[n//2:,n//2:]
        # Q4 = B10 - B00
        Q4 = B[n//2:,:n//2] - B[:n//2,:n//2]
        # Q5 = B11
        Q5 = B[n//2:,n//2:]
        # Q6 = B00 + B01
        Q6 = B[:n//2,:n//2] + B[:n//2,n//2:]
        # Q7 = B10 + B11
        Q7 = B[n//2:,:n//2] + B[n//2:,n//2:]

        # Then compute Mi = Pi*Qi by a recursive application of the function
        M1 = strassen(P1,Q1, m)
        M2 = strassen(P2,Q2, m)
        M3 = strassen(P3,Q3, m)
        M4 = strassen(P4,Q4, m)
        M5 = strassen(P5,Q5, m)
        M6 = strassen(P6,Q6, m)
        M7 = strassen(P7,Q7, m)

        # Following the recipe from the slides:

        C00 += M1 + M4 - M5 + M7
        C01 += M3 + M5
        C10 += M2 + M4
        C11 += M1 - M2 + M3 + M6

        return C
```

Python code A.8: strassen